Distributed Salt Authentication

Jake Walker

***Abstract* – Password management is a serious problem in modern web applications, the onus of which has always fallen on website owners and developers. But such a system is difficult to implement correctly, exposes a large surface for an attacker to exploit, and is vulnerable to brute force and dictionary attacks in the event of data compromise. In this paper, we propose a system that abstracts the responsibility of authentication management away from website owners, maintains an extremely small attack surface, and prevents password hash table disclosure in the event of a single system compromise.**

I. Introduction

Modern computer security has a fundamental flaw: every service is expected to secure its own data. Any attacker that compromises a single service exposes every customer of that service to potential fraud across all services in the ecosystem. The state of the internet is such that people often use the same usernames and passwords for multiple accounts, and compromising a single service is enough to compromise a large portion of an individual's online activity.

What we have is the worst of all worlds. There is a large attack surface available to attackers, complete with companies that do not have the expertise or the economic incentive to build robust security systems, and compromising any one system potentially compromises every other system.

We need a solution with the following characteristics:

1. Small attack surface
2. Abstracted responsibility of security management
3. Easy to use
4. No password exposure in the event of system compromise

In this paper, we propose a solution that satisfies these constraints: Distributed Salt Authentication. DSA requires that password salts and hashes be stored on specialized third party central servers, along with implementing a salt hashing system that prevents any one physical machine from knowing the true salt used to hash password values. Thus, an attacker that successfully compromises a system does not gain both the salt and the password hash necessary to carry out brute force and dictionary attacks on the password hash tables.

DSA abstracts the responsibility of authentication management away from website owners, maintains an extremely small attack surface, and prevents password hash table disclosure in the event of a single system compromise. It utilizes existing protocols and web infrastructure to guarantee secure data transfer, and website implementation would be as simple as copying a few lines of code to the website source.

II. Current state of security

Currently, every website owner is responsible for maintaining any information they collect from their users. Any site that would like to provide customer-specific content must require the user to set up an account, and must store that account information, including username and password, in a database somewhere. Once an attacker breaks into the system, recovering the plaintext password is a simple matter of power consumption, even if the website has implemented the correct security. If the attacker can correlate passwords with usernames, they can try these combinations at other websites, since username and password reuse is so widespread. Thus, an attacker that manages to hack the password table on an obscure web forum has likely gained access to significant credit card or bank account information for the majority of the forum users. The weakest link in the chain is all that must be broken in order to expose every online account a user owns.

Password Reuse

Ideally, customers simply *wouldn't* use the same username and password for multiple sites. There are good solutions in place for user-side

password management. Products like KeePass[1] and LastPass[2] abstract the difficulty of remembering different passwords for each different site, and are currently the only reasonable way for users to manage the hundreds of accounts they have. However, experience has told us that so far, the majority of users simply don't do this. It may be tempting to label this as user error, but user error is fundamentally the problem of the system designer. The internet, where every site you ever visit tries to get you to set up an account, is not constructed in such a way as to make security easy, nor does it force users to do security correctly. So long as there are users that follow bad security practice, this is a problem that developers are on the hook for- whether it's their fault or not.

Password Storage

The problem of password reuse can be mitigated by making the plaintext passwords difficult to recover, even once a system is penetrated. Companies with minimal security practice will store hashes of user passwords instead of the passwords themselves, so that a security breach doesn't automatically expose all of their user password data. Security conscious companies will go a step further, and store salted password hashes instead of unsalted password hashes, stopping the attacker from pre-computing hashes in a rainbow table attack. However, even when the company implements the correct security, they still store password hashes and salts on systems that expose a large attack surface. In the event that an attacker breaks in through any number of services running on the box, the attacker has access to both the hash and the salt. If the website has followed common guidelines for password hashing, including using SHA256 or SHA512 to hash, then brute forcing weak passwords is trivial given sufficient hardware[3]. Thanks to the advent of GPU clusters and cloud compute instance rentals, even many relatively "strong" passwords can be cracked with dictionary attacks in a matter of minutes.

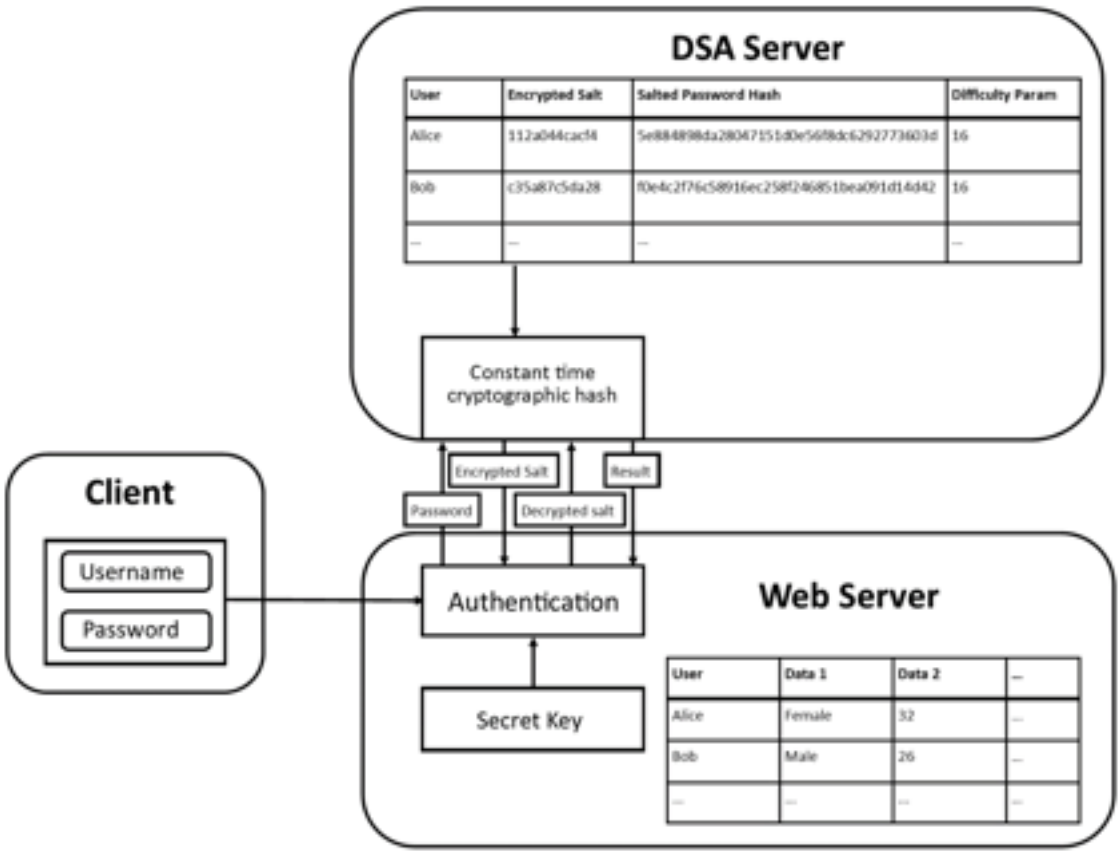
The problem is that standard hashes are designed to go *fast*. It's a feature. But in cryptography, the faster a hash goes, the easier it is to brute force. Key-stretching standards like bcrypt and PBKDF2 are hashing algorithms specifically designed with security in mind[4]. They allow for a "difficulty" parameter to be passed in along with the data to hash, and can therefore be scaled along with processor power. Processor time gets cheaper? No problem- increment the difficulty parameter, and computing the hash takes twice as many cycles. Using a cryptographically aware hashing algorithm can mitigate plaintext password recovery by attackers, and is always good security practice.

It is the case, however, that not every company implements good security. Horror stories abound of companies responding to password requests by emailing the user their original password. In a proper security system, the company should have **no idea** what your actual password is- they should only be storing a hash value. The lesson here is that once we decentralize the responsibility of security, people do it wrong. This is not surprising, and this will always be the case, as long as it's more economical for them to do so.

These flaws are not limited to small companies. Doing security right is hard, and doesn't have a clear and demonstrative link to profits, so lots of people just don't bother. Companies like nic.io[5], wantickets[6], PlentyOfFish[7], and lots of others[8] all screw this up.

III. Distributed Salt Authentication

As a solution to these problems, we propose a Distributed Salt Authentication system.



In DSA, we store encrypted salts and password hashes on a specialized third party DSA Server. In order to convert an encrypted salt into the true salt used, the DSA server sends the salt to the Web Server, which decrypts the encrypted salt using a secret private key, and sends the decrypted salt back to the DSA server. The DSA server uses this as the true salt to the password, which it then runs through its constant-time cryptographic hash with its own stored difficulty parameter.

DSA Workflow

When a user attempts to login, the Web Server passes the DSA server the submitted username and password information. The DSA replies with its stored encrypted salt value for the given user. The web server decrypts that salt with its own secret key, and passes the resulting true salt back to the DSA server. Thus, neither server has enough information to hash the password by itself.

If a secret key is ever discovered by an attacker, it must be revoked. Once a revocation is initiated, the DSA server iterates through its user list, passing the Web Server each encrypted salt value. The Web Server decrypts those values with the expired secret key to obtain the true salt, then encrypts the true salt with its new secret key, and replies to the DSA server with the new pre-hashed salt, which the DSA server stores in its table.

In this case, the DSA server has no choice but to rewrite the entire salt database. However, if an attacker gains access to a compromised web server and requests a secret key revocation, he doesn't gain any new information.

If an attacker has compromised the DSA server itself, the security procedure is the same. The only time there is a security threat is when an attacker knows both the secret key and the encrypted salt list. If either of these is leaked,

then revoking the key and reencrypting the salts is sufficient to render the attacker's information useless.

Web Server setup is as simple as copying a few lines of code to the website source and generating a secret key. To give true end-to-end password management, any DSA implementation should include a login handler that takes care of client-side password encryption and secure data transmission. A website owner should be able to place a few lines of javascript or php into their page and be up and running in a matter of minutes.

DSA Benefits

DSA has several advantages over traditional models. First, DSA abstracts the responsibility of authentication management away from the Web Server. The Web Server never writes the password, salt, or hash to disk, so any data breach on the server does not reveal user password hashes. Likewise, the DSA server does not have enough information to discover the true salt used with the password. Any attempt to brute force such a password + salt combo would require iterating over the entire address space of the hash size, which is computationally impractical for any existing hardware by several orders of magnitude. Ultimately, this mitigates the impact of Web Server security breaches to only stored user data, and will not allow attackers to parlay their access to a weakly protected site into access to a strongly protected site via brute force/dictionary attacks on password hashes. It also greatly mitigates the impact of DSA Server breaches. Once a breach is detected, the only cost to such a DSA Server intrusion would be the time you would need to spend renegotiating the encrypted salt values with the Web Server.

The only way for an attacker to gain access to the list of true salts would be to compromise both the DSA Server and the Web Server at the same time, since the private key can be changed and the encrypted salts regenerated. If an attacker did compromise both systems at the same time, he would be in exactly the same position an attacker is in today after compromising a system that has correctly implemented password security: He would have a list of salts and password hashes that have been implemented with a cryptographic hash function and a high difficulty parameter. The attacker would still need to implement non-trivial brute forcing and dictionary attacks on the hashed password list to obtain actionable information.

Because the password hashes and salts are stored on the DSA Server, the Web Server doesn't need to implement any password-specific storage itself. All it needs is to store a private key somewhere accessible to the functions implementing the web API calls. Further, the web server doesn't need to worry about proper housekeeping tasks like updating hash difficulty parameters for the password hashes as compute power increases. This both simplifies the life of the Web Server administrator, and centralizes the important, difficult-to-get-right aspects of password management. Much the same way you should never write your own hashing algorithm when there are numerous robust and well-tested solutions in existence, a Web Server administrator should never have to implement his own password storage.

Because the DSA server is specialized to perform this single function, the attack surface can be significantly reduced. The DSA server can run a custom operating system with limited port exposure and none of the extraneous code that tends to lead to security problems in a typical web server.

DSA Criticisms

If an attacker gains access to the DSA server, he doesn't care about encrypted hashes. He just has to wait for users to log in, and he gets their username/password combo for free.

If an attacker gains root access on DSA server, he can observe traffic and view the passwords of each user as they log in. Likewise, an attacker with root access to the Web Server can view the login attempts and results, and ultimately gain plaintext password access for any user that logs in while the machine is compromised. This is not unique to DSA; it is true whether DSA has been implemented or not. However, DSA does extend the attack surface for this particular type of exploit from just the Web Server to include the Web Server and the DSA Server. In practice, this is not nearly as big

of a problem as someone stealing millions of password hashes

Why bother with distributed salting? If you make the work factor high enough on bcrypt, it's basically impossible to brute force anyways.

At current prices[9], it costs roughly $200 to get the equivalent compute power of a 2.8 GHz processor running for a month. We could probably knock an order of magnitude off the price if we bought and ran specialized password hashing hardware ourselves. If we want our website to process 100 login attempts per second at peak times, we have to make a hash computable in .01 seconds on typical server hardware. Under these assumptions, an attacker could try about 260 million hashes for $20. If we assume a password of 6 lowercase alphanumeric characters, it's going to take us on average $36^6 / 2$ hashes, or roughly 1 Billion tries. So, roughly $80/password. That is not cheap. But that's also strict brute forcing, not dictionary attacks, which could make an attackers guesses a few orders of magnitude more effective. Even at $80 a pop, it might be worth it if you're targeting a specific username. It also might get cheaper to build high powered systems in the future. Of course, your web server might get faster too, but it's not clear which way we should bet on whether high powered cycle price growth will outpace low powered cycle price growth. Bottom line, we're not enough orders of magnitude above computability that it's infeasible for certain people to use this information; we still get some marginal utility out of implementing DSA instead of simple remote password storage with bcrypt.

Any website sufficiently large enough to matter is going to want to implement its own security.

Not necessarily. Certainly some of them will- Facebook is surely not turning over its security to an outside party anytime soon. But plenty of small- to mid-sized sites don't have dedicated teams of security engineers, and would gladly use a third party DSA system if it was easy and cheap. The target audience for DSA would be roughly the same as the target audience for Stripe[10].

If an attacker takes over the DSA server, he can generate login requests to the web service using the DSA username list and recover the true salt.

If an attacker gains root access to the DSA server for an indefinite amount of time, he can use this method to recover the true salt. The mitigation for this is twofold: the DSA client-side login handler must implement login throttling limits, and the DSA Server backend infrastructure must implement a solid intrusion detection system to catch compromised machines before they have time to implement this kind of attack.

In the end, *someone* needs to see both the true salt and the password at the same time in order to compute the password hash, and an attacker who manages to maintain a persistent and unnoticed presence on a compromised system is going to be seeing plaintext passwords as users log in anyways. The best defense against this type of attack is good intrusion detection systems on the DSA server- which, since it's centrally managed by security conscious engineering teams, is likely to be significantly better than similar IDS systems implemented by individual websites.

Why not just hash the password with the private key before sending it to the DSA server, instead of using distributed salt hashing?

The problem with this approach is that you can never revoke a compromised private key. Essentially, you've replaced the salt with a private key hash. But if the private key is ever compromised, you need the original password to hash with the new private key in order to rewrite the password hash in the database.

With DSA, the true salt remains the same; it is only the hashed salt value that needs to be updated. The Web Server simply has to decrypt the hashed salt with the old private key and encrypt it again with the new private key in order to update the salt value in the database. The password is not required.

Open Challenges

Distributed Salt Authentication is designed and particularly well suited for password management. However, it is unclear whether this functionality has use cases outside of this domain. Password management is unique in that you don't want to store the actual information the user has provided you- you just want to store a secret that is only discoverable if the user provides the required information a second time. Whether such a system could be expanded or modified to include storing other sensitive information like credit card numbers and social security numbers is an open question.

IV. Conclusion

By making every system admin secure their own data, we are multiplying the potential online attack surface not just by the number of website, but by the bad security practice of the lowest common denominator. Because of password and username reuse, online security is only as good as its weakest link- and its weakest link is barely trying. DSA solves the password reuse problem by significantly mitigating damage from successful attacks, and in particular by preventing even successful attackers from gaining access to password hash tables and implementing brute force and dictionary attacks to exploit password reuse. DSA is also easier for website owners to implement than traditional password storage solutions, while limiting their exposure to potential loss of customer information- and all the unpleasant legal and social consequences- in the event of a breach.

[1] http://keepass.com/

[2] https://lastpass.com/index.php

[3] http://codahale.com/how-to-safely-store-a-password/

[4] https://crackstation.net/hashing-security.htm

[5] http://andy.fine.io/2013/05/07/a-note-about-registering-io-domains.html

[6] http://blog.wxcs.com/2013/08/wantickets-com-stores-passwords-in-plain-text/

[7] http://techcrunch.com/2011/01/31/plentyoffish-ceo-we-were-hacked-almost-extorted-so-i-emailed-the-hackers-mom/

[8] http://plaintextoffenders.com/

[9] http://www.rackspace.com/cloud/sites/pricing/

[10] https://stripe.com/